
Semaphores

As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems. The first person who realized this years ago was **Edsger Dijkstra**, known among other things for his famous “shortest paths” algorithm in graph theory [D59], an early polemic on structured programming entitled “Goto Statements Considered Harmful” [D68a] (what a great title!), and, in the case we will study here, the introduction of a powerful and flexible synchronization primitive known as the **semaphore** [D68b,other]. Indeed, he invented this general semaphore before locks and condition variables; as you will see, one can use semaphores in both kinds of ways.

23.1 Semaphores: A Definition

A semaphore is as an object with an integer value that we can manipulate with two routines (which we will call `sem_wait()` and `sem_post()` to follow the POSIX standard). Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphore, we must first initialize it to some value, as this code below does:

In the figure, we declare a semaphore `s` and initialize it to the value of 1 (you can ignore the second argument to `sem_init()`

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

Figure 23.1: Initializing A Semaphore

for now; read the man page for details).

After a semaphore is initialized, we can call one of two functions to interact with it, `sem_wait()` or `sem_post()`¹. The behavior of these two functions is described here:

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting, wake 1
}
```

Figure 23.2: Initializing A Semaphore

For now, we are not concerned with the implementation of these routines, which clearly requires some care; with multiple threads calling into `sem_wait()` and `sem_post()`, there is the obvious need for managing these critical sections with locks and queues similar to how we previously built locks. We will now focus on how to *use* these primitives; later we may discuss how they are built.

A couple of notes. First, we can see that `sem_wait()` will either return right away (because the value of the semaphore was 1 or higher when we called `sem_wait()`), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into `sem_wait()`,

¹Historically, `sem_wait()` was first called P() by Dijkstra (for the Dutch word “to probe”) and `sem_post()` was called V() (for the Dutch word “to test”). Sometimes, people call them down and up, too.

and thus all be queued waiting to be woken. Once woken, the waiting thread will then decrement the value of the semaphore and return to the user.

Second, we can see that `sem_post()` does not ever suspend the caller. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes 1 of them up.

You should not worry here about the seeming race conditions possible within the semaphore; assume that the modifications they make to the state of the semaphore are all performed atomically (we will see how to implement them with locks and condition variables shortly).

23.2 Binary Semaphores (Locks)

We are now ready to use a semaphore. Our first use will be one with which we are already familiar: using a semaphore as a lock. Here is a code snippet:

```
sem_t m;
sem_init(&m, 0, X); // initialize semaphore to X; what should X be?

sem_wait(&m);
// critical section here
sem_post(&m);
```

Figure 23.3: A Binary Semaphore, a.k.a. a Lock

To build a lock, we simply surround the critical section of interest with a `sem_wait()/sem_post()` pair. Critical to making this work, though, is the initial value of the semaphore. What should it be?

If we look back at the definition of the routines `sem_wait()` and `sem_post()` routines above, we can see that the initial value of the semaphore should be 1. Imagine the first thread (thread 0) calling `sem_wait()`; it will first wait for the value of the semaphore to be greater than 0, which it is (the semaphore's value is 1). It will thus not wait at all and decrement the value

```
void *
child(void *arg) {
    printf("child\n");
    // signal here: child is done
    return NULL;
}

int
main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    // wait here for child
    printf("parent: end\n");
    return 0;
}
```

Figure 23.4: A Parent waiting for its Child

to 0 before returning to the caller. That thread is now free to enter the critical section. If no other thread tries to acquire the lock while thread 0 is inside the critical section, when it calls `sem_post()`, it will simply restore the value of the semaphore to 1 (and not wake any waiting thread, because there are no waiting threads).

The more interesting case arises when thread 0 holds the lock (i.e., it has called `sem_wait()` but not yet called `sem_post()`), and another thread (thread 1, say) tries to enter the critical section by calling `sem_wait()`. In this case, thread 1 will find that the value of the semaphore is 0, and thus wait (putting itself to sleep and relinquishing the processor). When thread 0 runs again, it will eventually call `sem_post()`, incrementing the value of the semaphore back to 1, and then wake the waiting thread 0, which will then be able to acquire the lock for itself.

In this basic way, we are able to use semaphores as locks. Because the value of the semaphore simply alternates between 1 and 0, this usage is sometimes known as a **binary semaphore**.

23.3 Semaphores As Condition Variables

Semaphores are also useful when a thread wants to halt its own progress waiting for something to change. For example, a thread may wish to wait for a list to become non-empty, so that it can take an element off of the list. In this pattern of usage, we often find a thread *waiting* for something to happen, and a different thread making that something happen and then *signaling* that it has indeed happened, thus waking the waiting thread. Because the waiting thread (or threads, really) is waiting for some **condition** in the program to change, we are using the semaphore as a **condition variable**. We will see condition variables again later, particularly when covering monitors.

A simple example is as follows. Imagine a thread creates another thread and then wants to wait for it to complete its execution (Figure 23.4).

When this program runs, what we would like to see here is the following output:

```
parent: begin
child
parent: end
```

The question, then, is how to use a semaphore to achieve this effect, and as it turns out, it is quite simple (Figure 23.5).

As you can see in the code, the parent simply calls `sem_wait()` and the child `sem_post()` to wait for the condition of the child finishing its execution to become true. However, this raises the question: what should the initial value of this semaphore be? (think about it here, instead of reading ahead)

The answer, of course, is that the value of the semaphore should be set to is the number 0. There are two cases to consider. First, let us assume that the parent creates the child but the child has not run yet (i.e., it is sitting in a ready queue but not running). In this case, the parent will call `sem_wait()` before the child has called `sem_post()`, and thus we'd like the parent to wait for the child to run. The only way this will happen is if the value of the semaphore is not greater than 0; hence,

```
sem_t s;

void *
child(void *arg) {
    printf("child\n");
    // signal here: child is done
    sem_post(&s);
    return NULL;
}

int
main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(c, NULL, child, NULL);
    // wait here for child
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

Figure 23.5: Parent waiting for its Child with Semaphores

0 as the initial value makes sense. When the child finally runs, it will call `sem_post()`, incrementing the value to 1 and waking the parent, which will then return from `sem_wait()` and complete the program.

The second case occurs when the child runs to completion before the parent gets a chance to call `sem_wait()`. In this case, the child will first call `sem_post()`, thus incrementing the value of the semaphore from 0 to 1. When the parent then gets a chance to run, it will call `sem_wait()` and find the value of the semaphore to be 1; the parent will thus decrement the value and return from `sem_wait()` without waiting, also achieving the desired effect.

23.4 The Producer/Consumer (or Bounded-Buffer) Problem

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value;    // line f1
    fill = (fill + 1) % MAX; // line f2
}

int get() {
    int tmp = buffer[use];   // line g1
    use = (use + 1) % MAX;   // line g2
    return tmp;
}
```

Figure 23.6: The Put and Get Routines

The final problem we will confront in this note is known as the **producer/consumer** problem, or sometimes as the **bounded buffer** problem, which was also first posed by Dijkstra [D72]. This problem is described in detail in the previous note on condition variables (Note ??).

Our first attempt at solving the problem introduces two semaphores, `empty` and `full`, which the threads will use to indicate when a buffer entry has been emptied or filled, respectively. The code for the `put` and `get` routines is in Figure 23.6, and our attempt at solving the the producer and consumer problem is in Figure 23.7.

In this example, the producer first waits for a buffer to become empty in order to put data into it, and the consumer similarly waits for a buffer to become filled before using it. Let us first imagine that `MAX=1` (there is only one buffer in the array), and see if this works.

Imagine again there are two threads, a producer and a consumer. Let us examine a specific scenario on a single CPU. Assume the consumer gets to run first. Thus, the consumer will hit line `c1` in the figure above, calling `sem_wait(&full)`. Because `full` was initialized to the value 0, the call will block the consumer and wait for another thread to call `sem_post()` on the semaphore, as desired.

```

sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line p1
        put(i);                     // line p2
        sem_post(&full);           // line p3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);           // line c1
        tmp = get();               // line c2
        sem_post(&empty);         // line c3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);   // ... and 0 are full
    // ...
}

```

Figure 23.7: Adding the Full and Empty Conditions

Let's say the producer then runs. It will hit line P1, calling `sem_wait(&empty)`. Unlike the consumer, the producer will continue through this line, because empty was initialized to the value MAX (in this case, 1). Thus, empty will be decremented to 0 and the producer will put a data value into the first entry of buffer (line P2). The producer will then continue on to P3 and call `sem_post(&full)`, changing the value of the full semaphore from 0 to 1 and waking the consumer (e.g., move it from blocked to ready).

In this case, one of two things could happen. If the producer

continues to run, it will loop around and hit line P1 again. This time, however, it would block, as the empty semaphore's value is 0. If the producer instead was interrupted and the consumer began to run, it would call `sem_wait(&full)` (line c1) and find that the buffer was indeed full and thus consume it. In either case, we achieve the desired behavior.

You can try this same example with more threads (e.g., multiple producers, and multiple consumers). It should still work, or it is time to go to sleep.

A Problem: Off To The Races

Let us now imagine that `MAX` is greater than 1 (say `MAX = 10`). For this example, let us assume that there are multiple producers and multiple consumers. We now have a problem: a race condition. Do you see where it occurs? (take some time and look for it) If you can't see it, here's a hint: look more closely at the `put()` and `get()` code. If you still can't see it, I bet you aren't trying. Come on, spend a minute on it at least.

OK, you win. Imagine two producers (Pa and Pb) both calling into `put()` at roughly the same time. Assume producer Pa gets to run first, and just starts to fill the first buffer entry (fill = 0 at line f1). Before Pa gets a chance to increment the fill counter to 1, it is interrupted. Producer Pb starts to run, and at line f1 it also puts its data into the 0th element of buffer, which means that the old data there is overwritten! This is a no-no; we don't want any data generated by a producer to be lost.

A Solution: Adding Mutual Exclusion

As you can see, what we've forgotten here is *mutual exclusion*. The filling of a buffer and incrementing of the index into the buffer is a critical section, and thus must be guarded carefully. So let's use our friend the binary semaphore and add some locks. Figure 23.8 shows our attempt.

Now we've added some locks around the entire `put()/get()` parts of the code, as indicated by the `NEW LINE` comments.

```

sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);           // line p0 (NEW LINE)
        sem_wait(&empty);          // line p1
        put(i);                     // line p2
        sem_post(&full);           // line p3
        sem_post(&mutex);          // line p4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);          // line c0 (NEW LINE)
        sem_wait(&full);           // line c1
        int tmp = get();           // line c2
        sem_post(&empty);          // line c3
        sem_post(&mutex);          // line c4 (NEW LINE)
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex = 1 because it is a lock (NEW LINE)
    // ...
}

```

Figure 23.8: Adding Mutual Exclusion (incorrectly)

That seems like the right idea, but it also doesn't work. Why? Deadlock. Why does deadlock occur? Take a moment to consider it; try to find a case where deadlock arises. What sequence of steps must happen for the program to deadlock?

Avoiding Deadlock

OK, now that you figured it out, here is the answer. Imagine two threads, one producer and one consumer. The consumer gets to run first. It acquires the mutex (line c0), and then calls `sem_wait()` on the full semaphore (line c1); because there is no data yet, this call causes the consumer to block and thus yield the CPU; importantly, though, the consumer still holds the lock.

A producer then runs. It has data to produce and if it were able to run, it would be able to wake the consumer thread and all would be good. Unfortunately, the first thing it does is call `sem_wait()` on the binary mutex semaphore (line p0). The lock is already held. Hence, the producer is now stuck waiting too.

There is a simple cycle here. The consumer *holds* the mutex and is *waiting* for the someone to signal full. The producer could *signal* full but is *waiting* for the mutex. Thus, the producer and consumer are each stuck waiting for each other: a classic deadlock.

Finally, A Working Solution

To solve this problem, we simply must reduce the scope of the lock. Here is the final working solution:

As you can see, we simply move the mutex acquire and release to be just around the critical section; the full and empty wait and signal code is left outside. The result is a simple and working bounded buffer, a commonly-used pattern in multi-threaded programs. Understand it now; use it later. You will thank me for years to come. Or not.

23.5 Reader-Writer Locks

Another classic problem stems from the desire for a more flexible locking primitive that admits that different data structure accesses might require different kinds of locking. For example, imagine a number of concurrent list operations, including

```

sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line p1
        sem_wait(&mutex);           // line p1.5 (MOVED THE MUTEX TO HERE)
        put(i);                     // line p2
        sem_post(&mutex);           // line p2.5 (... AND TO HERE)
        sem_post(&full);            // line p3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // line c1
        sem_wait(&mutex);           // line c1.5 (MOVED THE MUTEX TO HERE)
        int tmp = get();            // line c2
        sem_post(&mutex);           // line c2.5 (... AND TO HERE)
        sem_post(&empty);           // line c3
        printf("%d\n", tmp);
    }
}

```

Figure 23.9: Adding Mutual Exclusion (correctly)

inserts and simple lookups. While inserts change the state of the list (and thus a traditional critical section makes sense), inserts simply *read* the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to proceed concurrently. The special type of lock we will now develop to support this type of operation is known as a **reader-writer lock**. The code for such a lock is available here:

The code is pretty simple. If some thread wants to update the data structure in question, it should call the pair of operations `rwlock_acquire_writelock()` and `rwlock_release_writelock()`. Internally, these simply use the `writelock` semaphore to ensure that only a single writer can acquire the lock and thus enter

```
typedef struct _rwlock_t {
    sem_t writelock;
    sem_t lock;
    int readers;
} rwlock_t;

void rwlock_init(rwlock_t *lock) {
    readers = 0;
    sem_init(&lock, 0, 1); // binary semaphore
    sem_init(&writelock, 0, 1); // used to lock out a writer, or all readers
}

void rwlock_acquire_readlock(rwlock_t *lock) {
    sem_wait(&lock);
    readers++;
    if (readers == 1)
        sem_wait(&writelock);
    sem_post(&lock);
}

void rwlock_release_readlock(rwlock_t *lock) {
    sem_wait(&lock);
    readers--;
    if (readers == 0)
        sem_post(&writelock);
    sem_post(&lock);
}

void rwlock_acquire_writelock(rwlock_t *lock) {
    sem_wait(&writelock);
}

void rwlock_release_writelock(rwlock_t *lock) {
    sem_post(&writelock);
}
```

Figure 23.10: A Simple Reader-Writer Lock

the critical section to update the data structure in question.

More interesting is the `rwlock_acquire_readlock()` and `rwlock_release_readlock()` pair. When acquiring a read lock, the reader first acquires `lock` and then increments the `readers` variable to track how many readers are currently in-

side the data structure. The important step then taken within `rwlock_acquire_readlock()` occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling `sem_wait()` on the `writelock` semaphore, and then finally releasing the lock by calling `sem_post()`.

Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until *all* readers are finished; the last one to exit the critical section will call `sem_post()` on “writelock” and thus enable a waiting writer to acquire the lock itself.

This approach works (as desired), but does have some negatives, especially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers. More sophisticated solutions to this problem exist; perhaps you can think of a better implementation? Hint: think about what you would need to do to prevent more readers from entering the lock once a writer is waiting.

Finally, it should be noted that reader-writer locks should be used with some caution. They often add more overhead (especially with more sophisticated implementations), and thus do not end up speeding up performance as compared to just using simple and fast locking primitives [CB08]. Either way, they showcase once again how we can use semaphores in an interesting and useful way.

23.6 The Dining Philosophers

If I weren't lazy, I would write a bit about one of the most famous concurrency problems posed and solved by Dijkstra, known as the **dining philosopher's problem** [DHO]. However, I am lazy. The problem is famous because it is fun and somewhat intellectually interesting; however, its practical utility is low. I am a practical kind of guy, and thus have a hard time motivating the time spent to understand something that is so clearly academic. Look it up on your own if you are interested.

23.7 How To Implement Semaphores

Finally, let's use our low-level synchronization primitives, locks and condition variables, to build semaphores. It is fairly straightforward (see Figure 23.11).

```
typedef struct __Sem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Sem_t;

// only one thread can call this
void Sem_init(Sem_t *z, int value) {
    z->value = value;
    Cond_init(&z->cond);
    Mutex_init(&z->lock);
}

void Sem_wait(Sem_t *z) {
    Mutex_lock(&z->lock);
    while (z->value <= 0)
        Cond_wait(&z->cond, &z->lock);
    z->value--;
    Mutex_unlock(&z->lock);
}

void Sem_post(Sem_t *z) {
    Mutex_lock(&z->lock);
    z->value++;
    Cond_signal(&z->cond);
    Mutex_unlock(&z->lock);
}
```

Figure 23.11: Implementing Semaphores with Locks, CVs

As you can see from the figure, it is pretty simple. Just one lock and one condition variable, plus a state variable to track the value of the semaphore, is all you need.

23.8 Summary

Semaphores are a powerful and flexible primitive for writing concurrent programs. Some programmers use them exclusively, shunning simpler locks and condition variables, due to their great simplicity and utility.

In this note, we have presented just a few classic problems and solutions. If you are interested in finding out more, there are many other materials you can reference. One great (and free reference) is Allen Downey's book on concurrency [D08]. This book has lots of puzzles you can work on to improve your understanding of both semaphores in specific and concurrency in general. Becoming a real concurrency expert takes years of effort; going beyond what you learn in class is a key component to mastering such a topic.

References

- [D59] "A Note on Two Problems in Connexion with Graphs"
E. W. Dijkstra
Numerische Mathematik 1, 269271, 1959
Available: <http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf>
Can you believe people worked on algorithms in 1959? I can't.
- [D68a] "Go-to Statement Considered Harmful"
E.W. Dijkstra
Communications of the ACM, volume 11(3): pages 147148, March 1968
Available: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>
Sometimes thought as the beginning of the field of software engineering.
- [D68b] "The Structure of the THE Multiprogramming System"
E.W. Dijkstra
Communications of the ACM, volume 11(5), pages 341346, 1968
- [D72] "Information Streams Sharing a Finite Buffer"
E.W. Dijkstra
Information Processing Letters 1: 179180, 1972
Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF>
Did Dijkstra invent everything? No, but close. He even worked on one of the first systems to correctly provide interrupts!
- [D08] "The Little Book of Semaphores"
A.B. Downey
Available: <http://greentepress.com/semaphores/>
A great and free book about semaphores.
- [CB08] "Real-world Concurrency"
Bryan Cantrill and Jeff Bonwick
ACM Queue. Volume 6, No. 5. September 2008
Available: <http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=554>
- [DHO] "Hierarchical ordering of sequential processes"
E.W. Dijkstra
Available: <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>
The wikipedia page about this problem is also quite informative.